Performance Coupling: Case Studies for Improving the Performance of Scientific Applications

Jonathan Geisler Valerie Taylor

Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208

E-mail: {geisler,taylor}@ece.nwu.edu

Traditional performance optimization techniques have focused on finding the kernel in an application that is the most time consuming and attempting to optimize it. In this paper we focus on an optimization technique with a more global perspective of the application. In particular, we present a methodology for measuring the interaction, or *coupling*, between kernels within an application and describe how the measurements can be used to improve the performance of scientific applications. We discuss four case studies to demonstrate the use of this methodology. The first study involves the Conjugate Gradient benchmark from the NAS Parallel Benchmarks. The coupling measurement aided in the development of a new hybrid data structure and corresponding algorithm that slightly increased the performance of the program. The second study involves the Block Tridiagonal NAS Parallel Benchmark, for which the coupling parameter aided in revising the program to reduce the level-two cache misses by 14%. Next, we introduce improvements to an application in the SpecJVM benchmark suite resulting in 41% reduction in level-one cache misses. Lastly, we present results from the Seis application from the SPEChpc Benchmarks to illustrate the coupling parameters that may result from large-scale scientific applications.

Key Words: performance coupling, kernels, cache misses, NPB

1. INTRODUCTION

Traditional performance optimization techniques have focused on finding the kernel in a program that is the most time consuming and attempting to optimize it. An example of such an optimization entails restructuring an algorithm to increase data reuse (i.e., blocking), thereby reducing cache misses. It is well known that the performance increase that is achieved when optimizing a given kernel in isolation generally does not reflect the performance increase that occurs when the new kernel is included in the larger application [12]. This disparity in performance increase between the kernel and full application is due in part to a lack of understanding of how the interaction, or *coupling*, of kernels affects the performance of the application.

In this paper, we present a methodology for quantifying the coupling. In particular, we measure how kernels within an application perform in isolation in contrast to how kernels perform with nearby kernels in the application. The measurements lead to a parameter that is a ratio between the performance of the kernels executed together and the sum of the performance of the kernels executed in isolation. This parameter allows the algorithm designer to know how much a kernel interacts with other kernels in the application. With this ratio and a full understanding of the algorithm, the designer can improve the application performance by changing the kernel so that it can benefit from the work done by previous kernels and/or provide a benefit to succeeding kernels. By understanding where performance bottlenecks exist due to coupling, the algorithm designer will be able to develop more efficient applications that interact well between kernels, leading to overall improved performance.

This work focuses on single processor performance because of the importance of obtaining optimized serial codes. Amdahl's Law [1] emphasizes the importance of improving the performance of the serial portions to obtain an overall improvement in a parallel application. Furthermore, the SPMD programming methodology is based on using efficient sequential programs. The methodology generalizes to a parallel processing framework by adding another level of interaction to represent the interconnection network.

Within the single processor, we are looking at interaction that occurs within the memory hierarchy. With the increasing disparity between CPU and memory performance, it is important to optimize memory hierarchy usage. Also, the coupling parameter focuses on improved cooperation between kernels. Since the cache is designed to take advantage of sharing between different pieces of the same program, it is the first candidate for this methodology. The parameter, however, is not limited to measuring cache performance; the coupling parameter can be applied to other resources that are shared or reused (e.g., registers or networks).

In this paper we present four case studies that illustrate the usage of the coupling parameter. The coupling parameter measurements aided in the development of a new hybrid data structure and corresponding algorithm that slightly increased the performance of the Conjugate Gradient benchmark from the NAS Parallel Benchmarks. We achieved a reduction of 14% of level-two cache misses from the Block Tridiagonal benchmark from the same suite. We improved the _209_db benchmark from the SpecJVM suite by 41% less level-one cache misses. Finally, we demonstrated small improvements in the Seis application from the SPEChpc suite.

In the next section we describe the methodology of measuring coupling parameters for an application. Next, we give a simple example to demonstrate the coupling parameter in an easy to understand context. Then we present some benefits of using the coupling parameter through four case studies. Finally, we present related work and conclusions.

2. DESCRIPTION OF METHODOLOGY

This section describes our methodology and techniques for measuring coupling for both adjacent and non-adjacent kernels. We also include a description of the hardware architecture we got the results on.

2.1. Adjacent Coupling

The coupling parameter (c_{ij}) quantifies the interaction between adjacent kernels in an application. In this work, a kernel is a unit of computation that denotes a logical entity within the larger context of an application. The unit may be a loop, procedure, or file depending on the level of granularity of detail that is desired from the measurements.

To compute the parameter c_{ij}, three measurements must be taken:

1. p_i is the performance of kernel i alone,
2. p_j is the performance of kernel j alone, and
3. p_{ij} is the performance of kernels i and j assuming kernel i immediately preceeds kernel j in the application.

These measurements are done in the same way as determined by the application. For example, if the application looks like Algorithm 2.1, then we would execute Algorithm 2.2, to measure p_C, Algorithm 2.3 to measure p_D, and Algorithm 2.4 to measure p_{CD}. Similarly, we would execute Algorithm 2.5 to measure p_E and Algorithm 2.6 to measure p_{DE}.

Algorithm 2.1 Example Application

```
kernel A
for $i \leftarrow 1$ to 25 do
  kernel B
  for $j \leftarrow 1$ to 10 do
    kernel C
    kernel D
  end for
end for
kernel E
kernel F
```

Algorithm 2.2 Measure p_C

```
for $i \leftarrow 1$ to 25 do
  for $j \leftarrow 1$ to 10 do
    kernel C
  end for
end for
```

The three measurements above can improve the performance of an application by monitoring the sharing or reuse of any resource to which the application has access. Example resources that might be measured are different levels of the mem-

Algorithm 2.3 Measure p_D

```
for $i \leftarrow 1$ to 25 do
  for $j \leftarrow 1$ to 10 do
    kernel D
  end for
end for
```

Algorithm 2.4 Measure p_{CD}

```
for $i \leftarrow 1$ to 25 do
  for $j \leftarrow 1$ to 10 do
    kernel C
    kernel D
  end for
end for
```

Algorithm 2.5 Measure p_E

```
kernel E
```

Algorithm 2.6 Measure p_{DE}

```
for $i \leftarrow 1$ to 25 do
  for $j \leftarrow 1$ to 10 do
    kernel D
  end for
end for
kernel E
```

ory hierarchy or processor instruction slots on multithreaded machines. This work looks at the cache usage by measuring misses at the various levels.

It should be noted that interactions between all pairs of kernels is *not* necessary. The value c_{ij} represents the direct interaction between two adjacent kernels in an application (i.e., in the sense of the control flow of the application). Hence, for each unique application control path that involves N kernels, only $N - 1$ pairwise interactions are measured.

In general, the value c_{ij} is equal to the ratio of the measured performance of the pair of kernels to the expected performance resulting from combining the isolated performance of each kernel. For the case of cache misses we expect p_{ij} to be the sum of p_i and p_j if there is no interaction between kernels. Since c_{ij} is the measurement of interaction between kernels, we compute it as the ratio of the actual performance of the kernels together to that of no interaction: $c_{ij} = \frac{p_{ij}}{p_i + p_j}$. The summation of the isolated performance is not applicable to all performance metrics, such as floating point operations per second (flop/s); a weighted average would be used in this case. Different performance metrics may require different mathematical formulations for combining the isolated performance.

We group the parameters into three sets:

- $c_{ij} = 1$ indicates no interaction between the two kernels, yielding no change in performance.
- $c_{ij} < 1$ results from some resource(s) being shared between the kernels, producing a performance gain (i.e., constructive coupling). This occurs when the two kernels have lower cache misses when run together than separately.
- $c_{ij} > 1$ occurs when the kernels interfere with each other, resulting in a performance loss (i.e., destructive coupling). This occurs when the two kernels have higher cache misses when run together than separately.

Therefore, it should be the goal to use code that minimizes c_{ij} to achieve best performance.

2.2. Non-adjacent Coupling

Previously, we discussed kernels that occur consecutively in the call graph of an application. These kernels may be said to be *adjacent* and one would expect them to greatly affect each other. *Non-adjacent* kernels are separated by one or more kernels and can have coupling. In Section 4.3, we present results that indicate measurable interaction between non-adjacent kernels.

Non-adjacent kernels can exhibit coupling despite the intermediate kernels that execute for several reasons:

1. The intermediate kernels are small and do not affect the state of the shared resource.
2. The itermediate kernels have a high coupling with the first kernel and so they do not make much difference in the state of the resource.
3. The shared resource is so large that it can handle the needs of multiple kernels without degrading performance. The level-two cache is a good example of this since it can hold data structures as large as 4 Mbytes. The improvement in BT relied on this fact to fit two data structures in the level-two cache by relocating them in the cache.

Recall the initial definition of the coupling parameter was

$$c_{ij} = \frac{performance\ together}{combined\ isolated\ performances} \tag{1}$$

The key to reformulating the equation to account for non-adjacent kernels is to keep the ratio semantics the same. The numerator is the performance of the kernels run together, and the denominator is the performance one would expect to result if there was no interaction between the kernels. The new coupling parameter retains this relationship in the following manner:

We define S as the set of kernels to be measured. We measure the performance of the kernels independently (p_k for every kernel $k \in S$), and the performance of the kernels together (p_S) to compute the coupling parameter (c_S). Finally, we must define some function $\mathcal{F}$ such that $\underset{k \in S}{\mathcal{F}}(k)$ defines the performance of the kernels with no interaction. The equation becomes

$$c_S = \frac{p_S}{\underset{k \in S}{\mathcal{F}}(k)} \tag{2}$$

In the case of cache misses, we expect kernels with no interaction to perform as the sum of the individual kernels, so we set $\mathcal{F} = \Sigma$. Thus the equation for the coupling parameter when measuring the interactions of kernels for the cache is $c_S = \frac{p_S}{\sum_{k \in S} p_k}$. If we reconsider the two adjacent kernels case, we set $S = \{i, j\}$, and the following hold true:

- $p_S \equiv p_{ij}$
- $\sum_{k \in S} p_k = p_i + p_j$
- $c_S \equiv c_{ij}$

Therefore, we have an equation that retains the original definition but expands to multiple non-adjacent kernels.

2.3. Machine Description

For all of our experiments, we used the SGI Origin2000 at Northwestern University in the Center for Parallel and Distributed Computing. It is an eight way symmetric multiprocessor, but we only use one processor. Each processor is a 64 bit chip running at 195 MHz capable of 390 Mflop/s (2 flop/s per cycle) [10]. It has 64 floating point registers and 64 integer registers. The machine has separate level-one instruction and data caches, but a unified level-two cache. Each level one cache is 32 Kbytes with two-way set associativity. The instruction cache has a line size of 64 bytes, while the data cache line size is 32 bytes. The 4 Mbyte level-two cache is two-way set associative with a 128 byte line size. The total main memory is 1 Gbyte in size.

2.4. Kernel Measurements

Each of the applications were divided into an initialization kernel and computation kernels. The initialization kernel contained all the code needed to set up the correct data values for the computational kernels. The structure of the benchmarks analyzed in this paper is such that the initialization section occurs once, outside of the loop around the computational kernels.

The number of cache misses of the initialization kernel (p_{init}) in isolation was measured using the performance counters in hardware. Next, the number of cache misses for the initialization kernel and the computational kernel i (p_{init+i}) was measured since the execution of the kernel often depends on the data being valid (e.g., random floating point values would produce NaNs and hardware traps not found in any run of the code). The value for p_i was then computed as $p_i = p_{init+i} - p_{init}$.

To get an accurate measurement of the cache misses, we measured p_i, p_j, and p_{ij} inside the loop that surrounds the computational section. The cache misses for p_i consists of the initialization kernel followed by a loop containing only kernel i. This is consistent with what is done when a kernel is taken out of an application and optimized. The same applies to measuring the cache misses for p_j. In the case of p_{ij}, the measurement consists of the initialization kernel followed by a loop containing kernel i and kernel j. This measurement strategy does not contain any cache flushes, which can artificially produce kernels with coupling values of 1.0.

3. SIMPLE EXAMPLE

To illustrate the ideas behind the coupling parameter and verify the model, we use a synthetic program with two kernels that generate specified data streams. The synthetic program consists of two kernels that accesses memory with a stride of one. For all the programs in this section, we used code to implement the pseudocode in Algorithm 3.1.

Algorithm 3.1 Simple Example pseudocode

```
$sizeA \leftarrow$ size of first array
$sizeB \leftarrow$ size of second array
$offsetB \leftarrow$ second array start - first array start
for $i \leftarrow 1$ to 400 do
    kernel 1:
    for $j \leftarrow 1$ to $sizeA$ do
        touch memory at location $j$
    end for
    kernel 2:
    for $j \leftarrow 1$ to $sizeB$ do
        touch memory at location $j + Boffset$
    end for
end for
```

It is very easy to predict the number of misses because we know the exact access patterns for the program and the cache characteristics of the machine, which can be combined to form a prediction in the following manner. For arrays smaller than the cache, the number of misses is the number of misses needed to pull the array into the cache, or $\frac{array\ size}{cache\ line\ size}$, and for arrays larger than the cache, each cache line must be refetched because it was flushed by another array location, thus the total number of misses are $\frac{number\ of\ accesses}{cache\ line\ size}$.

To illustrate the predictability of the synthetic program, we ran a number of experiments to measure the first level cache misses for the kernels with array sizes of 4 Kbytes to 512 Kbytes. The runs ran with $sizeB$ from Algorithm 3.1 set to 0, so only kernel 1 was measured. Each array element is four bytes. The results are given in Table 1 for array sizes larger than or equal to 32 Kbytes. The hardware counters do not generate interrupts until at least 2,053 cache misses occur, and the predicted number of misses for the array sizes smaller than 32 Kbytes is less than 2053.

The predicted column is computed as described previously. The last column is the relative error of the predicted value. By observing the final column, we can see that the synthetic program performs as predicted. The only case where the program does not perform as predicted is 32 Kbytes. The reason for this is a small, but noticeable system overhead. The prediction expects the entire array to stay in the cache, but due to looping overhead, hardware interrupts, and other applications running (e.g., daemons), this prediction is overly optimistic. These additional overheads consist of 1,408 bytes per iteration, resulting in 44 misses per iteration or 17,600 misses during four hundred iterations. This was confirmed with two independant tests.

After measuring each kernel in isolation, experiments were conducted to measure the synthetic program, involving the two kernels, with various array sizes. A coupling value is generated for each run. We use three sets of experiments to illustrate the concept of the coupling parameter. The first experiment considers the kernels accessing arrays with the same starting address ($Boffset = 0$), but different sizes ($sizeA \neq sizeB$). The last two experiments demonstrate how the coupling parameter changes when the starting address of the arrays are changed. The measurements performed for verification, given in Table 1, are used as the isolated values (p_i and p_j) in the coupling parameter calculations for the experiments. Tables 2-5 contain the results of combining the kernels, represented as the misses attributed to each kernel. The value of p_{ij} is the misses for the entire algorithm.

The results from the first experiment are given in Tables 2 (a) and (b). The first column identifies the kernel pair. The second column gives the cache misses for the first kernel, and the third column for the second kernel. The two columns can be compared to the measured values in Table 1 to identify if the cache misses increased (destructive coupling) or decreased (constructive coupling) when the two kernels were executed together as compared to when the two kernels were executed individually.

In Table 2 (a), the misses for kernel two (128 Kbytes) decreased as compared to Table 1; however, the misses for kernel one (sized 4 Kbytes through 32 Kbytes) increased. When executed in isolation, the first kernel is able to load the entire array into the cache during the first iteration, resulting in no misses during the remaining iterations. When the first kernel is executed with the second kernel, the first kernel must reload the data into the cache during each iteration because the second kernel flushes the cache every iteration, causing destructive coupling. Conversely, the second kernel when run in isolation always reloads the array into the cache. When run with the first kernel, however, the second kernel can access the array that the

TABLE 1

Level one cache misses

Array Size	Predicted	Measured	Relative Error
512 Kbytes	6,553,600	6,555,229	-0.00024850
256 Kbytes	3,276,800	3,276,588	0.00006470
128 Kbytes	1,638,400	1,640,347	-0.00118694
64 Kbytes	819,200	817,094	0.00257742
32 Kbytes	1,024	18,477	-0.94457974

TABLE 2a

Kernels exhibiting coupling

Kernels	Kernel 1 misses	Kernel 2 misses	Coupling Parameter
4 Kbytes ⇒ 128 Kbytes	57,484	1,582,863	1.0
8 Kbytes ⇒ 128 Kbytes	626,165	1,014,182	1.0
16 Kbytes ⇒ 128 Kbytes	427,024	1,211,270	0.99874843
32 Kbytes ⇒ 128 Kbytes	254,572	1,412,464	1.00495049

TABLE 2b

Kernels not exhibiting coupling

Kernels	Kernel 1 misses	Kernel 2 misses	Coupling Parameter
64 Kbytes ⇒ 128 Kbytes	819,147	1,638,294	1.0
128 Kbytes ⇒ 128 Kbytes	1,638,294	1,638,294	0.99874843
256 Kbytes ⇒ 128 Kbytes	3,280,694	1,634,188	0.99958246
512 Kbytes ⇒ 128 Kbytes	6,553,176	1,642,400	1.0

first kernel has already loaded into the cache causing a decrease in the number of misses, or constructive coupling. In this example, the coupling parameter remains at 1.0 because the constructive coupling (the size of kernel one's array reused by kernel two) is exactly the same as the destructive coupling (the size of kernel one's array that must be reloaded into the cache), thereby cancelling each other. Hence, the coupling parameter measures effective performance. In Table 2 (b), the misses for both kernels remained about the same as the isolated execution given in Table 1. Therefore, as expected, the coupling parameter is approximately 1.0.

The second experiment explores the impact of moving the starting address of the array such that the data left in the cache by the second kernel is immediately accessed by the first kernel on the next iteration ($-sizeB < Boffset < 0$). We expect this alignment to produce constructive coupling by reducing the number of misses of the first kernel. The results are given in Table 4.

In Table 4, the misses for kernel one (sized 64 Kbytes through 512 Kbytes) decreased as compared to Table 1. When run in isolation, kernel one was forced to reload its array into the cache during each iteration. By reusing the data left in the cache by kernel two, kernel one does not incur as many cache misses causing constructive coupling. Since there is no destructive coupling to offset the constructive coupling, the coupling parameter is less than 1.0. As the size of kernel one increases, however, the coupling parameter increases from 0.85380116 to 0.95591182, since the amount of reuse become a smaller percentage of the total number of accesses.

The coupling values below 1.0 for the second experiment is very significant. We have not altered either kernel to change the number of cache misses when run in isolation. The only reason for the decrease in cache misses in kernel one is due to kernel two loading the data that kernel one needs. This coupling can only be measured when they are executed together.

The last experiment also looks at the impact of moving the starting address of the array that the first kernel accesses; the address, however, is set such that the data accessed by the two kernels do not overlap ($Boffset > sizeA$). This "misalignment" can only cause destructive coupling by causing extra misses, since the two kernels will never share data.

In Table 5, the misses for kernel one increased as compared to Table 1. Unlike when run in isolation, kernel one is forced to reload all of its data into the cache during each iteration causing destructive coupling. In addition, the second kernel continues to reload all of its data into the cache during every iteration. Since there is no constructive coupling to balance the destructive coupling, the coupling parameter is greater than 1.0 and increases as the size of kernel one increases since kernel one must reload more data as its size increases.

The coupling values above 1.0 for the third experiment is another significant result. Once again, we have not altered either kernel to produce more misses when run in isolation. The only reason for the increase in cache misses is due to the interference produced when both kernels are run together. When coupling the results of the previous experiment and this experiment together, we can see that both constructive and destructive coupling can occur and that the only way to measure its existence is to measure the performance of kernels together.

4. CASE STUDIES

In this section we describe four examples for which the coupling parameter was used to make modifications with the applications that resulted in performance improvements. The first example improves the level-one cache misses while the second improves the level-two cache misses. The third examines coupling when the kernels that interact do not occur in succession. Finally, the last study looks at an industrial sized application.

4.1. NPB: Conjugate Gradient

The Conjugate Gradient benchmark from the NAS Parallel Benchmarks [2] solves the equation $Ax = b$ using an iterative process to search through the solution space. The main computational complexity occurs during the matrix-vector multiply section of the code. We give results for the S class ($N = 1400$, and 102,200 nonzeros), but achieve similar results for the W class ($N = 7000$, and 637,000 nonzeros). Both sizes take fifteen iterations to converge to the correct solution.

We divide the Conjugate Gradient benchmark into three kernels:

1. *Initialization* sets up the data structures after randomly generating the nonzero elements in the sparse matrix.
2. *Matrix-Vector Multiplication* performs the largest piece of computation that can have varying numbers of cache misses based on the data structure that stores the matrix.
3. *Remaining Vector Operations* performs the rest of the computations needed to perform the conjugate gradient application and then any cleanup after the application is finished.

We considered three different sparse matrix representations in addition to the original representation for the matrix-vector multiply kernel:

- CMNS[3] stores the matrix non-zeros by columns.
- SPAR[17] is similar to CMNS, but stores some zeros to facilitates vector processing.
- ITPACK[5] stores a compressed matrix whose size is determined by row with the largest number of nonzeros. It stores zeros for the rows smaller than the largest row.
- The original representation[2] stores the matrix non-zeros by rows.

The results of measuring the isolated performance of each kernel and the coupling parameters are given in Figure 1. The node values are the level-one cache misses and the edge values are the coupling parameters. Initial inspection of Figure 1 immediately rules out the ITPACK representation as the best algorithm because of its poor performance (nearly four times worse than the other three algorithms), which is not compensated by a small enough coupling parameter. This leaves CMNS, SPAR, and the original representation as possibilities. CMNS and SPAR have destructive coupling with the rest of the code ($c_{ij} > 1.0$), whereas the original code has negligible impact on the rest of the code ($c_{ij} = 1.02$). If one considers cache misses only, which is the traditional method, one would select the original representation or CMNS. The coupling values provide information that can lead to new insights as described below.

TABLE 4

Kernels exhibiting constructive coupling (Exp. 2)

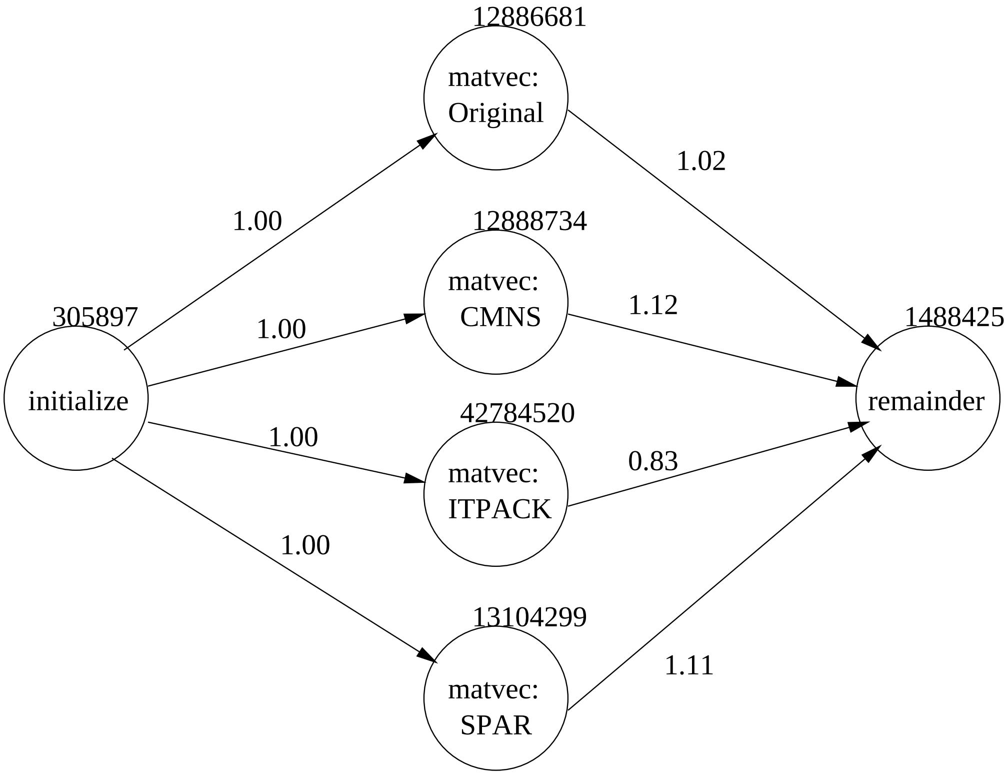
Kernels	Kernel 1 misses	Kernel 2 misses	Coupling Parameter
64 Kbytes ⇒ 128 Kbytes	457,819	1,640,347	0.85380116
128 Kbytes ⇒ 128 Kbytes	1,276,966	1,640,347	0.88923654
256 Kbytes ⇒ 128 Kbytes	2,917,313	1,642,400	0.92734864
512 Kbytes ⇒ 128 Kbytes	6,193,901	1,640,347	0.95591182

TABLE 5

Kernels exhibiting destructive coupling (Exp. 3)

Kernels	Kernel 1 misses	Kernel 2 misses	Coupling Parameter
4 Kbytes ⇒ 128 Kbytes	53,378	1,636,241	1.03003754
8 Kbytes ⇒ 128 Kbytes	102,650	1,640,347	1.06257822
16 Kbytes ⇒ 128 Kbytes	203,247	1,642,400	1.12515644
32 Kbytes ⇒ 128 Kbytes	410,600	1,640,347	1.23638613

FIG. 1. Conjugate gradient coupling graph



Next, we studied the code carefully to understand why the various coupling parameters existed. First, we examined ITPACK since it had such a good coupling parameter between the matrix-vector multiply and the remaining vector operations. For ITPACK, the $\vec{w}$ vector is strided through during each iteration of the outside loop, which corresponds to different columns of the compressed sparse matrix. In the CG code however, the first vector used by the remaining vector operations is the $\vec{w}$ vector (the resultant vector of the matrix-vector multiply operation). ITPACK and CMNS force the $\vec{w}$ vector to be left in the cache causing fewer cache misses in the vector updates code. The other data structures only iterate through the $\vec{w}$ vector once (original code) or in a non-strided manner (SPAR). The problem with ITPACK, however, is that it accesses too many zero entries in the compressed sparse matrix, causing a significant number of cache misses.

In the CG code, the last data structure used by the remaining vector operations before the matrix-vector multiply is the $\vec{p}$ vector (the multiplier vector of the matrix-vector multiply operation). Both ITPACK and the original code access $\vec{p}$ in a non-strided manner that benefits from having $\vec{p}$ in the cache. CMNS and SPAR only iterate through the $\vec{p}$ vector once, causing only the first few accesses of $\vec{p}$ to hit the cache before $\vec{p}$ is flushed by other accesses in the matrix-vector multiplication.

The above characteristics suggest a code for matrix-vector multiply such that the first part uses $\vec{p}$ frequently (to take advantage of $\vec{p}$ being left in the cache by the remainder of the code) and the latter part strides through $\vec{w}$ to leave $\vec{w}$ in the cache to be used by the remainder of the code. This was achieved by splitting the sparse matrix, A, in half. The first $N/2$ columns of the matrix were stored row-wise, using the original data structure. The second $N/2$ columns of the matrix were stored column-wise using CMNS. Hence the hybrid data structure does not store any zero entries.

The new algorithm resulted in 12,898,999 misses for the matrix-vector multiply and a coupling parameter of 1.0. The number of misses is in the range of original and CMNS, but the coupling is better. The reduction in coupling for the new algorithm is enough such that the new algorithm has 17,451 fewer total cache misses (1% less) than the original data structure.

Also, we explored the effect of the location of the split in matrix A on the coupling parameter. We studied six splits and the coupling parameters associated with them. The results are in Table 6. The split identifies the fraction of the matrix that is stored row-wise followed by the fraction of the matrix that is stored column-wise. As can be seen, the 15/16,1/16 split produces the lowest coupling parameter and the 31/32,1/32 split produces the largest. The coupling reduces as the fraction of the matrix is stored row-wise. This makes sense since only a few columns need to be retrieved to put the $\vec{w}$ vector into the cache for the remainder of the code while many rows may take advantage of the $\vec{p}$ vector being left in the cache. The 15/16,1/16 split provides this advantage best. This split reduces the number of cache misses by 197088, or 1.3% over the original code.

4.2. NPB: Block Tridiagonal

The Block Tridiagonal Benchmark (BT) from the NAS Parallel Benchmarks [2] uses an implicit algorithm to solve the 3-D compressible Navier-Stokes equations. In the BT benchmark, the flux Jacobians are fully diagonalized resulting in block-tridiagonal matrices. The x, y, and z dimensions have been decoupled, so they are solved separately. The computation proceeds in a two phase manner. The first phase calculates the right hand side consisting of local difference stencils only. The second phase involves solving 5×5 systems of equations and multiplying 5×5 matrices as part of a specialized Gaussian elimination solver. According to [11], there is significant data reuse in the BT kernel.

We divided the application into seven kernels:

1. *Initialization* sets all the initial values for the various matrices, vectors, and scalars. It reads the input file to respond to user requests. Finally, it probes the system for run-time values like number of processors.
2. *Copy Faces* work done is the phase one computation of the right hand side.
3. *X Solve* solves the problem in the x dimension.
4. *Y Solve* solves the problem in the y dimension.
5. *Z Solve* solves the problem in the z dimension.
6. *Add* performs a matrix update.
7. *Final Cleanup* verifies the solution integrity and cleans up any data structures along with printing out the results of the computation.

The coupling values for the second level cache using the smallest dataset ($12 \times 12 \times 12$ that converges in sixty iterations) are in Table 7. The Copy Faces ⇒ X Solve coupling parameter for the second level cache was a high value of 4.92. The two kernels used approximately 406 Kbytes and 2509 Kbytes in data structures. In total, this is less than the 4096 Kbytes available from the level-two cache. Thus, the extra misses occurring at level-two are conflict misses–not capacity misses. By printing the addresses of the data structures used in the two kernels, we were able to determine which ones conflicted with each other. Finally, we changed the declaration section so the conflicting arrays had different stack addresses. This modification required no changes to the algorithm. The effect of changing the declarations is similar to that of padding an array without the disadvantage of adding extra memory.

Interestingly, after the modification the number of misses for the Copy Faces kernel isolated increased from 2,096 to 16,637 and the number of misses of the X Solve kernel isolated increased from 12,576 to 32488, but the number of misses for the coupled kernels dropped from 72,181 to 27772, or 62%. When run with the full application, we saw a 14% improvement in level-two misses. The change made a dramatic improvement, but would have been rejected if judged solely on its effect on the individual kernels. This is because of the number of cache hits X Solve got from Copy Faces with the new variable layout.

Because of the three dimensional layout of the matrix, the second largest coupling occurs for Y Solve ⇒ Z Solve. The layout helps the coupling at X Solve ⇒ Y Solve, but seems to hurt the coupling for Y Solve ⇒ Z Solve. Further research is needed to develop a data structure that gives good cache performance for such three dimensional data structures.

With both CG and BT, the amount of time spent studying and improving the code took less than a day because the coupling parameter pointed directly to the problem area. The improvements presented here were easy to develop with such a directed focus given by the coupling parameter.

4.3. SpecJVM: _209_db

The Standard Perfomance Evaluation Corporation (SPEC) has produced a set of eight benchmarks for Java Virtual Machines. It was produced in 1998 to provide a standard measurement critera for different Java platforms [15]. This benchmark is used to simulate the performance one would expect to receive when implementing a full database in Java. The program reads a starting database of 1 Mbyte and performs actions on that database that it gets from a command file. During the execution of the application, 224 Mbytes of objects are allocated. The benchmark performs nine operations that we identified as basic kernels:

- *Add* inserts a new record to the end of the database and sets the current pointer to reference the newly added record.
- *Begin* sets the current pointer to the first record.
- *Delete* deletes the record that the current pointer references.
- *End* sets the current pointer to the last record.
- *Find* locates a record with specific criteria and sets the current pointer to reference it.
- *Modify* changes the values of the record referenced by the current pointer.
- *Next* sets the current pointer to the record following the currently referenced record. This operation does not perform any actions if the current pointer references the last record.
- *Previous* sets the current pointer to the record preceeding the currently reference record. This operation does not perform any actions if the current pointer references the first record.
- *Sort* arranges the records according to a user-specified field. This operation does not perform any actions if the records are already sorted on the same field and no changes have been made to the database since the last sort.

Certain operations may require other operations to perform correctly (e.g., Find performs the Sort operation).

The benchmark performs these opertions in a specific order to approximate what a real database would see. We created a synthetic load that performed a much simpler task by minimizing the amount of interactions between kernels. The synthetic load performed each operation in sequence leading to eight adjacent interactions since there were nine total kernels (e.g., Add preceeded Find and followed Previous, leading to two adjacent interactions). We recognized the synthetic load did not fully represent the typical benchmark input, but thought it would give a first approximation to where some inefficiencies existed in the code. We performed the coupling measurements and they all came out ≤ 1.07. The results are in Table 8. When applying Formula 2 for $S =$ all kernels, we got $c_S = 2.44587598$ for L1 and $c_S = 3.33148572$ for L2, so we knew there was interaction in the synthetic workload not being measured.

We began to measure non-adjacent kernels with the results in Table 9. This indicated End and Delete were having poor coupling since the coupling values dramtically increased when considering both kernels. We began to look at the code to identify the source of the problem. The End operation is extremely small in source code and appears to provide little overhead. However, the two operations

TABLE 6

Coupling parameters for various hybrid splits

fraction of matrix	coupling for		
row-wise	column-wise	Init ⇒ MVM	MVM ⇔ Remain
1/4	3/4	0.99952666	1.00443131
1/2	1/2	1.00048015	0.98511904
3/4	1/4	1.00015651	0.96764705
7/8	1/8	1.00046019	0.98688046
15/16	1/16	1.00030266	0.92162554
31/32	1/32	1.0	1.36863823

TABLE 7

Coupling parameters for BT Level 2 cache

Kernels	$p_i + p_j$	p_{ij}	c_{ij}
Copy Faces ⇒ X Solve	14,672	72,181	4.91964285
X Solve ⇒ Y Solve	31,964	24,235	0.75819672
Y Solve ⇒ Z Solve	45,457	80,827	1.77809798
Z Solve ⇒ Add	26,724	24,366	0.91176470
Add ⇒ Copy Faces	2,751	3,537	1.28571428
Add ⇒ Final	3,537	2,620	0.74074074

TABLE 8

Coupling for synthetic workload of _209_db

Kernels	L1 coupling	L2 coupling
Begin + Next	0.79411764	0.50310559
Next + End	0.73333333	0.33789954
End + Previous	0.56250000	0.34304207
Previous + Add	0.90588235	0.90069084
Add + Find	1.06298366	1.01060985
Find + Delete	0.98484395	0.98568333
Delete + Modify	0.97701745	1.04830965
Modify + Sort	1.01337928	1.01375173

share a function (`set_index()`) to make sure the records are indexed properly. Further study discovered that the Java datatypes used to store the record contained indexing functions, resulting in the `set_index` call being redundant. We removed the redundant indexing in the code and relied on the indexing internal to the Java data structure and got the following results: the L1 cache misses decreased by 65% and the L2 cache misses decreased by 73%. To verify that the modifications made a difference on the originally specified workload, we measured the performance of the original workload and achieved these results: the L1 cache misses decreased by 41% and the L2 cache misses decreased by 25%.

4.4. SpecHPC: Seis

SpecHPC contains several industrial sized applications to measure high-end computing systems. Seis is a data processing application for seismic data [7]. This application comes with many different predefined dataset sizes and is included to illustrate the coupling values that result in an industrial application. The benchmark required significant disk capacity exceeding that available on the experimental system. Therefore, Seis was only run with the “tiny” dataset available. This dataset generates between 20 and 100 Mbytes of trace files while processing.

The results for the Seis application are given in Table 10. The first column gives the coupled kernels. The second column is the sum of the misses for the isolated kernels. The next column is the number of misses for the coupled kernels, and the final column is the computed coupling parameter. The coupling parameters for Seis are striking. Only one indicates constructive coupling, and over 75% indicate very bad destructive coupling. The two highest coupling parameters correspond to kernels not originally developed for the benchmark (VRFY and RATE). Both were added by SPEC for verification and measurement purposes respectively. It can be observed that these kernels do not dominate the execution time and we can assume that SPEC was not concerned with getting the best performance from these new kernels. Another factor affecting the coupling parameters for Seis is the granularity differences between it and the NPB. Seis uses file level granularity while NPB use procedure level granularity. The change in granularity may be a factor in the destructive coupling since most files do not share much data. The sharing typically occurs within file units.

We created a graph like Figure 2 for each kernel. Each bar represents a kernel being added to the string of kernels run. The height of the bar is the number of misses in DGEN when that kernel is added to the kernels. The first bar is the number of misses in DGEN when VSBF is run. Obviously, there are not very many misses in DGEN since none of its kernels are run (some misses do occur during initialization accounting for a non-zero value). The second bar is the number of misses when GEOM is added so that both VSBF and GEOM are running. Again, very little misses are measured. The third bar has a large jump because that is when DGEN is added to the kernels run (now VSBF + GEOM + DGEN). This correlates to the number of misses in DGEN run in isolation. Notice the number of misses in DGEN when FANF is added. It is larger than any other bar. This indicates that FANF has negative coupling with DGEN. This is confirmed by the measurements in Table 10. The bar is lower for later kernels indicating that FANF's destructive coupling is overcome by constructive coupling by the other kernels. We

TABLE 9

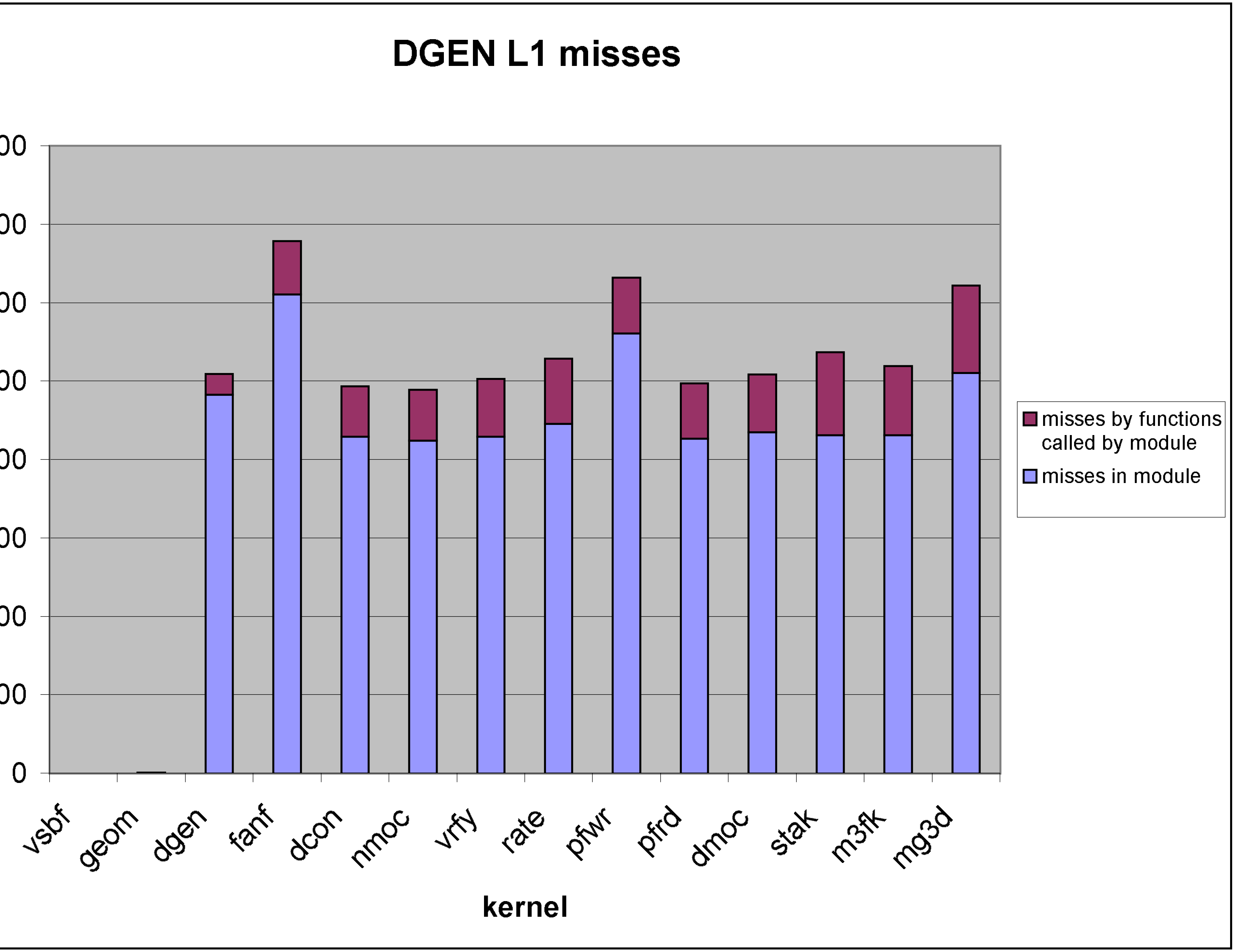
Non-adjacent coupling

Kernels	L1 coupling	L2 coupling
End + Previous + Add + Find	2.97032450	0.98574802
Previous + Add + Find + Delete	2.41850009	1.03921043
Add + Find + Delete + Modify	2.32512692	1.02006764
Find + Delete + Modify + Sort	2.50096665	0.97766960
End + Previous + Add + Find + Delete	4.59245463	7.66919864
Previous + Add + Find + Delete + Modify	1.05384983	1.05741370
Add + Find + Delete + Modify + Sort	1.01668185	1.00578967
End + Previous + Add + Find + Delete + Modify	5.02989208	7.66002490
Previous + Add + Find + Delete + Modify + Sort	1.02090840	1.01790079

TABLE 10

Seis level-one coupling parameters

Kernels	$p_i + p_j$	p_{ij}	c_{ij}
VSBF ⇒ GEOM	45,166	129,339	2.86363636
GEOM ⇒ DGEN	585,105	829,412	1.41754385
DGEN ⇒ FANF	562,522	1,328,291	2.36131386
FANF ⇒ DCON	192,982	800,670	4.14893617
DCON ⇒ NMOC	322,321	443,448	1.37579617
NMOC ⇒ PFWR	158,081	484,508	3.06493506
PFWR ⇒ VRFY	36,954	394,176	10.66666666
VRFY ⇒ RATE	34,901	566,628	16.23529411
RATE ⇒ PFRD	131,392	650,801	4.95312500
PFRD ⇒ DMOC	3,436,722	3,208,839	0.93369175
DMOC ⇒ STAK	4,459,116	4,496,070	1.00828729
STAK ⇒ M3FK	1,233,853	3,845,269	3.11647254
M3FK ⇒ MG3D	336,223,916	343,312,925	1.02108419


DGEN L1 misses
00
00
00
00
00
00
00
00
0
vsbf
geom
dgen
fanf
dcon
nmoc
vrfy
rate
pfwr
pfrd
dmoc
stak
m3fk
mg3d
kernel
misses by functions called by module
misses in module

explored the destructive coupling with FANF and DGEN so that we could remove it and take advantage of the constructive coupling with the other kernels.

Study of FANF and DGEN concluded that the FFTs (forward and reverse) that FANF performs were pushing the data out of cache that DGEN was using. In general, this data movement cannot be changed because the FFTs are critical to the FANF algorithm. FANF, however, was not performing the FFT in an optimal manner for cache use.

When FANF is called, it is given a pointer to the data it needs. It then calls the forward FFT and points to a temporary location which the FFT should return the values in. The FFT, then copies the data to a third location to perform the work before copying the results to the destination location. FANF then performs a filter operation in the temporary location before calling the reverse FFT. The result location of the reverse FFT is the location of the real data, but before the data is copied there, the reverse FFT is performed in a fourth temporary location. This is shown in Figure 3.

By performing as much of the FANF work inplace as possible instead of using temporary arrays, we were able to improve the number of L1 cache misses of FANF by over 18% and the overall application by over 1%. Much of Seis is designed similarly to FANF and performing a similar optimization to all the kernels should improve the performance of the application by 5-10%.

5. RELATED WORK

Allan Snavely defined a similar concept as symbiosis [14]. His work studies the interaction of two separate programs on a multithreaded machine instead of interaction within an application. The equations used to quantify interaction uses a similar ratio of isolated and simultaneous execution times. His work in [13] indicates that dynamic scheduling of multiple highly tuned applications still produces better efficiency on the Tera MTA. Our work focuses, however, on improving the performance of a single application.

Rafael Saavedra did much work characterizing various benchmarks [8] by decomposing them into high-level Fortran statements. He then counted the number of times each statement occurred in the execution of the program. By measuring the execution time of each statement on various target machines, he was able to predict the total execution time of the benchmarks by multiplying the statement execution times by the number of times it occurred and then summing that product over all statements. Rafael's work demonstrated that measurements of high level constructs can result in accurate predictions (over 50% of the predictions had less than 10% error) if cache or compiler optimizations are not used.

In [9], he added terms in his model to account for cache effects. This improved the accuracy of his model to more than 80% of the predictions with less than 10% error. Our work complements Rafael's work in quantifying and understanding the interaction between kernels.

Many metrics already exist to measure performance of serial and parallel programs. The most important is execution time. Some metrics measure specific operations, such as instructions, cache misses, or execution pipeline stalls. These metrics do not relate the operations from one kernel to another, however. Others, such as speedup, sizeup [16], and measured serial fraction [4] attempt to relate some theoretical performance to the observed performance, but they are based on the full program resulting in one value per application; hence coupling is not represented. While the coupling parameter does relate a theoretical performance to an observed performance (kernels in isolation summed together vs. kernels run together), a single application is represented by multiple coupling parameters (one for each kernel interaction).

Larry Carter et. al. have studied the performance impacts of hierarchical tiling [6]. Their technique focuses on improving a single kernel within an application, however the additional information that the coupling parameter provides indicates that the technique would be useful across kernels. The coupling parameter can indicate which cross-kernel tilings should be pursued and which should be ignored.

6. SUMMARY

In this paper, we defined the coupling parameter for adjacent kernels ($c_{ij} = \frac{p_{ij}}{p_i+p_j}$) and generalized it to non-adjacent kernels ($c_S = \frac{p_S}{\underset{k\in S}{\mathcal{F}}(k)}$). This allowed us to quantify the interaction between kernels in an application. We measured the coupling parameter for four applications for the first two levels of the memory hierarchy. Using the coupling parameter, we were able to develop a new data structure for the conjugate gradient benchmark that improves level one cache misses 1%. Also, we improved the level-two cache misses for BT by 14%. Next, we studied the effects of coupling on non-adjacent kernels. _209_db provided 41% improvement when removing the redundant data structure. Finally, Seis indicated that a large range of coupling values exists on industrial applications with a small improvement demonstrated.

The improvements above were achieved through various means:

- Improved data structure for CG resulted in better interaction between the matrix-vector multiply and the remaining vectore operations.
- Changed memory location in BT reduced the level-two conflict misses without changing the algorithm.
- Removed redundant operations in the Java code to take advantage of the values stored internally by the native data structure.
- Optimized the cache usage of seis by removing temporary storage to reduce cache pollution by the temporaries.

All of these improvements were pointed to by the coupling parameter. Without the assistance of the coupling parameter, these improvements would have ranged from not investigated due to no known problem (CG and _209_db) to too difficult to diagnose (Seis). In all four cases, the coupling parameter quickly led us to hone on to an inefficiency that could be improved.

REFERENCES

1. G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS 1967 Spring Joint Computer Conference*, pages 483 – 485, April 1967.
2. David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA, December 1995.

3. I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

4. Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

5. D.R. Kincaid, J.R. Respess, D.M. Young, and R.G. Grimes. ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software*, 8:302–322, 1982.

6. Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 1998.

7. Charles C. Mosher and Siamak Hassanzadeh. ARCO seismic processing performance evaluation suite: Seis 1.0 users's [sic] guide. Technical report, ARCO Exploration and Technology and Sun Microsystems, October 1993.

8. Rafael H. Saavedra and Alan Jay Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report CSD-92-715, University of California, Berkeley, 1992.

9. Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect on benchmark run times. Technical Report CSD-93-767, University of California, Berkeley, 1993.

10. Subhash Saini and David Bailey. Hot chips for high performance computing. In *SuperComputing Tutorials*, November 1996.

11. William Saphir, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.1 results. Technical Report NAS-96-010, NASA, August 1996.

12. Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical Report GIT-CC-94/38, Georgia Institute of Technology, September 1994.

13. Allan Snavely and Larry Carter. Symbiotic jobscheduling on the Tera MTA. In *Proceedings of Third Workshop on Multi-Threaded Execution, Architecture, and Compilers*, January 2000.

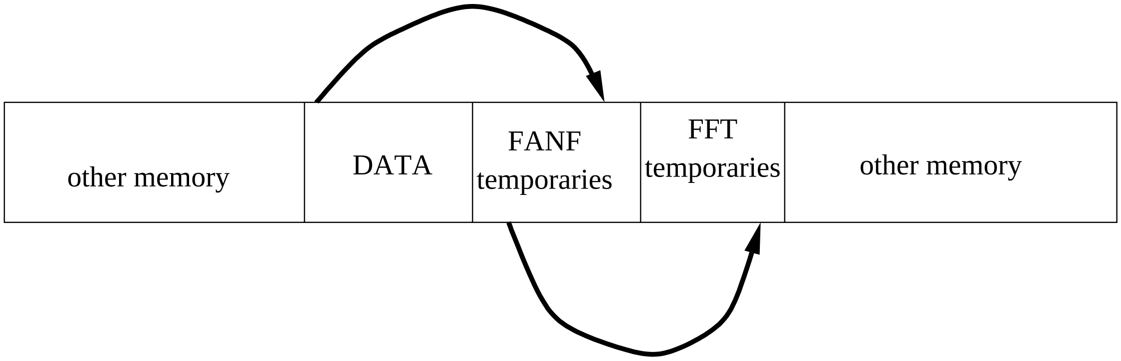
14. Allan Snavely, Nick Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. Explorations in symbiosis on two multithreaded architectures. In *Proceedings of Second Workshop on Multi-Threaded Execution, Architecture, and Compilers*, January 1999.

15. SPEC, Inc. Results at http://www.spec.org. 2000.

16. Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, 1991.

17. Valerie E. Taylor, Abhiram Ranade, and David G. Messerschmitt. SPAR: A new architecture for large finite element computations. *IEEE Transactions on Computers*, 44(4):531–545, April 1995.

FIG. 3. Data movement in FANF


other memory
DATA
FANF temporaries
FFT temporaries
other memory